

An $N \log N$ Parallel Fast Direct Solver for Kernel Matrices

Chenhan D. Yu*, William B. March[†] and George Biros[§]

^{*}Department of Computer Science

^{††} Institute for Computational Engineering and Science

The University of Texas at Austin, Austin, Texas, USA

^{*}chenhan@cs.utexas.edu, [†]march@ices.utexas.edu, [†]gbiros@acm.org

Abstract—Kernel matrices appear in machine learning and non-parametric statistics. Given N points in d dimensions and a kernel function that requires $\mathcal{O}(d)$ work to evaluate, we present an $\mathcal{O}(dN \log N)$ -work algorithm for the approximate factorization of a regularized kernel matrix, a common computational bottleneck in the training phase of a learning task. With this factorization, solving a linear system with a kernel matrix can be done with $\mathcal{O}(N \log N)$ work. Our algorithm only requires kernel evaluations and does *not* require that the kernel matrix admits an efficient global low rank approximation. Instead our factorization only assumes low-rank properties for the *off-diagonal blocks* under an appropriate row and column ordering. We also present a hybrid method that, when the factorization is prohibitively expensive, combines a partial factorization with iterative methods. As a highlight, we are able to approximately factorize a dense $11M \times 11M$ kernel matrix in 2 minutes on 3,072 x86 “Haswell” cores and a $4.5M \times 4.5M$ matrix in 1 minute using 4,352 “Knights Landing” cores.

I. INTRODUCTION

Let \mathcal{X} be a set of N points $\in \mathbb{R}^d$ and let $\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ be a given kernel function. The *kernel matrix* is the $N \times N$ matrix whose entries are given by $K_{ij} = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$ for $i, j = 1, \dots, N$, $\mathbf{x}_i, \mathbf{x}_j \in \mathcal{X}$.

Kernel matrices appear in unsupervised and supervised statistical learning, Gaussian process, regression, and non-parametric statistics [8], [12], [27], [33]. Solving linear systems with kernel matrices is an algebraic operation that is required in many kernel methods. The simplest example is ridge regression in which we solve $\lambda I + K$, where $\lambda > 0$ is a regularization parameter that controls generalization accuracy and I is the identity matrix. This linear solve can be prohibitively expensive for large N because K is typically *dense*. For example, consider the Gaussian kernel,

$$\mathcal{K}(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{1}{2} \frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{h^2}\right), \quad (1)$$

where h is the *kernel bandwidth*. For small h , K approaches the identity matrix whereas for large h , K approaches the rank-one constant matrix. The first regime suggests sparse approximations while the second regime suggests global low-rank approximations. But for the majority of h values, K is neither sparse nor globally low-rank. Direct factorization of $\lambda I + K$ requires $\mathcal{O}(N^3)$ work, whereas a Krylov iterative method costs $\mathcal{O}(N^2)$ work per iteration

and may require 1000s of iterations. This *complexity barrier* has limited the use of kernel methods for large-scale problems [6], [20].

Contributions. We exploit *hierarchically low-rank* approximations in which we assume that K can be approximated well by $D + UV$, where D is block-diagonal and the U and V matrices have low rank. Using such a decomposition, we improve the factorization algorithm presented in [36]. In that paper the block-diagonal plus sparse decomposition was done using ASKIT¹, a method introduced in [21], [22] (see §II-A). ASKIT approximates K in $\mathcal{O}(dN \log N)$ time and the algorithm in [36] factorizes the ASKIT approximation in $\mathcal{O}(N \log^2 N)$ time (see §II-B). Roughly speaking, ASKIT is based on the approximation of K as the sum of a block-diagonal matrix and a low-rank matrix followed by recursion for each diagonal block. We refer this process as the construction of the *hierarchical representation* of K . Once we have this representation, we can factorize K by applying recursively the Sherman-Morrison-Woodbury (SMW) formula. The factorization has to be done for different values of λ during cross-validation studies. Therefore optimizing the factorization is crucial for the overall performance of a kernel method. In this paper, we extend the factorization scheme presented in [36] in several ways:

- We present an algorithm that factorizes the ASKIT approximation in $\mathcal{O}(N \log N)$ time instead of $\mathcal{O}(N \log^2 N)$ and we demonstrate its performance on several datasets (see §II-C and §V).
- We present a hybrid *level-restricted* factorization scheme that reduces dramatically the factorization time by using a Krylov iterative solver on a much smaller system than the original (see §II-C). The new method can be used with matrices for which [36] fails.
- We study performance on Intel’s “Knights Landing” (KNL) architecture. We introduce an optimized matrix-free kernel summation that reduces the storage requirements of the factorization without having a very significant impact on wall-clock time (see §II-D).

In our numerical experiments, we measure the performance of the method on several different datasets. ASKIT has been applied to polynomial, Matern, Laplacian, and Gaussian

¹ Approximate Skeletonization Kernel Independent Treecode. The ASKIT library is available at <http://padas.ices.utexas.edu/libaskit>.

kernels in arbitrary dimensions. Due to space limitations we only present the factorization results for the Gaussian kernel function. Also the Gaussian kernel is among the hardest to compress in high dimensions. We examine the performance of the method for different bandwidth ranges that are relevant to learning tasks, its sensitivity to the regularization parameter λ , its performance as we increase the number of points, and its numerical stability (see §III).

Limitations. Not all kernel matrices admit a good hierarchical low-rank decomposition. Typically this is related to the intrinsic dimensionality of the dataset at different scales. So ASKIT and subsequently our method can fail. If ASKIT can compress the matrix, the second potential point of failure is the choice of regularization parameter. If it is too small, our algorithm (as well as [36]) can become numerically unstable. We can numerically detect the instability, but it is not clear how to fix it while maintaining the log-linear complexity of the algorithm. However, small regularization often results in poor learning performance so this corner case is not important in applications. We discuss this in more detail in §III and §V. Also our methods cannot be applied to hierarchical decompositions in which D is sparse and not just block diagonal. For such decompositions, our method can be used as a preconditioner, as discussed in [36].

Related work. Nystrom methods and their variants [7], [13], [28], [34] can be used to build fast factorizations. However, not all kernel matrices can be approximated well by Nystrom methods [15], [18], [23], [31]. Factorization methods based on hierarchical decomposition have been studied for kernel matrices from points in two or three dimensions [1], [2], [4], [9], but less so in high dimensions with a few exceptions [15], [36]. Early works discussing parallel operations for hierarchical matrices on shared memory system include bulk synchronous parallelization [16] and DAG-based task parallelism [17]. Distributed factorization and operations were discussed in [14], [32]. The difficulties of generalizing low-dimensional factorizations in high-dimensions are discussed in [21], [36].

II. METHODS

We begin with a sketch of hierarchical matrices and direct solvers in §II-A. We also briefly summarize the ASKIT algorithm which we use as the basis for our new methods. We describe parallel factorization schemes in §II-B and highlight the novelty of our approach over [36]. We then introduce our hybrid iterative/direct solver in §II-C.

A. Hierarchical Matrices and Treecodes

Broadly speaking, we consider a matrix $K \in \mathbb{R}^{N \times N}$ to be *hierarchical* if it can be partitioned as

$$K = \begin{bmatrix} K_{11} & K_{1r} \\ K_{r1} & K_{rr} \end{bmatrix} = \begin{bmatrix} K_{11} & 0 \\ 0 & K_{rr} \end{bmatrix} + \begin{bmatrix} 0 & K_{1r} \\ K_{r1} & 0 \end{bmatrix}. \quad (2)$$

where the *off-diagonal* blocks K_{1r} and K_{r1} can be accurately *approximated* by a low-rank factorization and the

on-diagonal blocks K_{11} and K_{rr} are themselves hierarchical. Note that the low-rank structure is *not invariant* on permutations, it very strongly depends on the ordering of the columns (or rows since the matrix is symmetric). For notational convenience we write $K \approx \tilde{K} = D + UV$, where U and V are rank s and D is also hierarchical, where we use \tilde{K} to indicate the approximate kernel matrix.

Inverting hierarchical matrices. When K admits this hierarchical low-rank approximation, then we can efficiently approximate K^{-1} using the Sherman-Morrison-Woodbury formula along with recursion:

$$\begin{aligned} \tilde{K} &= D + UV = D(I + WV), \text{ and } W = D^{-1}U. \\ \tilde{K}^{-1} &= (I + WV)^{-1}D^{-1} \\ &= (I - W(I + VW)^{-1}V)D^{-1} \\ &= (I - WZ^{-1}V)D^{-1}, \text{ where } Z = I + VW. \end{aligned} \quad (3)$$

Recursion is used to invert D^{-1} . After obtaining D^{-1} we compute $W = D^{-1}U$ for a rank- s matrix U and factorize the smaller reduced system $Z = I + VW \in \mathbb{R}^{s \times s}$. The scheme can be easily extended to invert $\lambda I + K$.

To turn this formulation into an algorithm, we need (1) a method to partition K so that off-diagonal blocks have low-rank, (2) an efficient way to compute the low-rank factors U and V , and (3) a scheme to construct the inverse. For the first two tasks we use ASKIT, a method we recently developed [21]–[23]. ASKIT uses geometric information (the input points) to permute K by partitioning the points recursively using a binary tree. Interactions between points in a treenode correspond to diagonal blocks of K . In the recursion, the children of the node can be used to define the block partitioning of the parent block, similar to (2). Next, we summarize ASKIT features that are necessary for this paper. Please see [23] for the complete details on ASKIT.

Partitioning the matrix. We use a ball tree [26] to partition \mathcal{X} . Starting with the root node (which contains the entire data set), nodes are partitioned into two children (with an equal number of points) by a splitting hyperplane. This recursive splitting terminates when a node has less than m points, a user-specified parameter. The root has *level* $l = 0$ and the leaves $l = \mathcal{D} = \log_2(N/m)$, the *depth* of the tree. In the following, we overload α, β to indicate both *binary tree nodes* and the *indices of the points* that belong to these nodes; $|\alpha|$ is the number of points in α ; and l, r indicate the left and right children of the node. We define $\mathcal{X} \in \mathbb{R}^{d \times N}$ to be the matrix of all points and \mathcal{X}_α to be the points owned by tree node α , (i.e., $\mathcal{X}_\alpha = \{\mathbf{x}_i | \forall i \in \alpha\}$).

Computing low rank approximations. Let α be the points in a leaf node. Let $S = \{1, \dots, N\} \setminus \alpha$. The *skeletonization* of a node α is a rank- s approximation of $K_{S\alpha}$ using s columns of $K_{S\alpha}$. We refer to these columns as the *skeleton* of α , denoted by $\tilde{\alpha}$. Skeletonization is done using the Interpolative Decomposition (ID) [11]. Using a pivoted rank-revealing QR factorization, the ID finds $\tilde{\alpha}$ and $P_{\tilde{\alpha}\alpha} \in \mathbb{R}^{s \times |\alpha|}$ such that

$$K_{S\alpha} \approx K_{S\tilde{\alpha}}P_{\tilde{\alpha}\alpha}. \quad (4)$$

Algorithm II.1 $[\tilde{\alpha}, P_{\tilde{\alpha}\alpha}] = \text{Skeletonize}(\alpha)$

if α **is leaf** **then return** $[\tilde{\alpha}, P_{\tilde{\alpha}\alpha}] = \text{ID}(\alpha)$;
 $[\tilde{1}, \cdot] = \text{Skeletonize}(1)$; $[\tilde{\mathbf{r}}, \cdot] = \text{Skeletonize}(\mathbf{r})$;
return $[\tilde{\alpha}, P_{\tilde{\alpha}[\tilde{1}\tilde{\mathbf{r}}]}] = \text{ID}([\tilde{1}\tilde{\mathbf{r}}])$;

The first s pivots from the QR define $\tilde{\alpha}$. Using the QR, we can compute $P_{\tilde{\alpha}\alpha} = K_{S\tilde{\alpha}}^\dagger K_{S\alpha}$. This scheme however results in $\mathcal{O}(dN^2m)$ complexity for the overall factorization. We can turn it to a $\mathcal{O}(d \log Nm)$ scheme by sampling a small subset S' of S and using it instead of S [23]. The approximation rank s is chosen such that $\sigma_{s+1}(K_{S'\alpha})/\sigma_1(K_{S'\alpha}) < \tau$, where τ is user-specified and σ are the singular values estimated by the diagonal of the rank-revealing QR.

For a non-leaf α , we first compute the skeletons $\tilde{1}$ and $\tilde{\mathbf{r}}$ of the children of α and then we compute the skeleton $\tilde{\alpha} \subset \tilde{1} \cup \tilde{\mathbf{r}} = [\tilde{1}\tilde{\mathbf{r}}]$ and the projection matrix $P_{\tilde{\alpha}[\tilde{1}\tilde{\mathbf{r}}]}$ using another ID decomposition (Algorithm II.1). Once the skeletonization of every node (but the root) is computed, we can compute the i_{th} entry of Kw by $K_{i\cdot}w \approx K_{i\alpha}w(\alpha) + \sum_{l=0}^D K_{i\tilde{\beta}}P_{\tilde{\beta}\tilde{\beta}}w(\beta)$, where $i \in \alpha$ and β are the siblings of α and its ancestors.

In ASKIT and [36], U and V of a node α for (3) are

$$\begin{bmatrix} 0 & K_{1\mathbf{r}} \\ K_{\mathbf{r}1} & 0 \end{bmatrix} \approx U_\alpha V_\alpha = \begin{bmatrix} & K_{1\tilde{\mathbf{r}}} \\ K_{\mathbf{r}\tilde{1}} & \end{bmatrix} \begin{bmatrix} P_{11} & \\ & P_{\mathbf{r}\mathbf{r}} \end{bmatrix}. \quad (5)$$

In this work we use the fact that $K_{1\mathbf{r}} = K_{\mathbf{r}1}^T$. Thus, $K_{1\mathbf{r}}$ can be approximate in two equivalent forms: $K_{1\tilde{\mathbf{r}}}P_{\mathbf{r}\mathbf{r}}$ or $P_{11}K_{\mathbf{r}\tilde{1}}$. Here we use the second form to write

$$\begin{bmatrix} 0 & K_{1\mathbf{r}} \\ K_{\mathbf{r}1} & 0 \end{bmatrix} \approx U_\alpha V_\alpha = \begin{bmatrix} P_{11} & \\ & P_{\mathbf{r}\mathbf{r}} \end{bmatrix} \begin{bmatrix} K_{1\tilde{\mathbf{r}}} \\ K_{\mathbf{r}\tilde{1}} \end{bmatrix}. \quad (6)$$

We will see that having the P terms on the left allows us to design an $\mathcal{O}(N \log N)$ factorization algorithm.

Level restriction. We provide details on the level-restriction feature of ASKIT. As we saw in Algorithm II.1, the skeletonization proceeds in a bottom-up traversal of the ball tree. As we traverse the tree, the off-diagonal blocks are growing larger and, depending on the problem, the necessary rank s can increase to the extent that no compression takes place. To guarantee accuracy, skeletonization of α should terminate if $\tilde{\alpha} = \tilde{1} \cup \tilde{\mathbf{r}}$. In some of our numerical experiments, instead of using this criterion, we use L to represent the level at which the skeletonization stops.

With level restriction, the factorization described in [36] *cannot be used*. We introduce a hybrid iterative/direct scheme that addresses this shortcoming in §II-C.

B. Fast direct solver

We have sketched how we compute the UV approximations in \tilde{K} using ASKIT. We now discuss how to use them in the context of (3) to solve $\lambda I + K$ directly. For simplicity, we describe the case where $\lambda = 0$, but all the algorithms we describe trivially generalize to the $\lambda \neq 0$ case. We first consider the case in which no level restriction takes place.

We assume that all the internal nodes have been skeletonized. The factorization of \tilde{K} proceeds using a bottom-up traversal of the tree. At the leaf level, we factorize

$K_{\alpha\alpha}^{-1} \in \mathbb{R}^{m \times m}$ using LAPACK's GETRF. For an internal node α , we need U_α , W_α , V_α , and Z_α^{-1} . Using (3), we can write out $\tilde{K}_{\alpha\alpha} = D_\alpha(I + W_\alpha V_\alpha)$ as

$$\begin{bmatrix} \tilde{K}_{11} & \\ & \tilde{K}_{\mathbf{r}\mathbf{r}} \end{bmatrix} \left(I + \begin{bmatrix} \hat{P}_{1\tilde{1}} & \\ & \hat{P}_{\mathbf{r}\tilde{\mathbf{r}}} \end{bmatrix} \begin{bmatrix} K_{\tilde{\mathbf{r}}1} & K_{\tilde{\mathbf{r}}\mathbf{r}} \end{bmatrix} \right), \quad (7)$$

where we define $\hat{P}_{1\tilde{1}} = \tilde{K}_{11}^{-1}P_{11}$ and $\hat{P}_{\mathbf{r}\tilde{\mathbf{r}}} = \tilde{K}_{\mathbf{r}\mathbf{r}}^{-1}P_{\mathbf{r}\mathbf{r}}$ (notice the “hat” notation). Therefore, $\hat{P}_{\alpha\tilde{\alpha}}$ requires “inverting” $\tilde{K}_{\alpha\alpha}^{-1}$, which in turn requires traversing all the descendants of α (the subtree rooted at α) and recursively applying (7). (We introduced this scheme in [36] and results in $\mathcal{O}(dN \log^2 N)$ complexity.) But as we will see shortly this subtree traversal is not necessary.) Once we have W_α , we use the SMW formula to invert $(I + W_\alpha V_\alpha)^{-1}$. This inverse requires $W_\alpha Z_\alpha^{-1} V_\alpha$, which in terms of the block decomposition of α , can be written as

$$\begin{bmatrix} \hat{P}_{1\tilde{1}} & \\ & \hat{P}_{\mathbf{r}\tilde{\mathbf{r}}} \end{bmatrix} \begin{bmatrix} I & K_{\tilde{\mathbf{r}}1}\hat{P}_{\mathbf{r}\tilde{\mathbf{r}}} \\ K_{\tilde{\mathbf{r}}1}\hat{P}_{1\tilde{1}} & I \end{bmatrix}^{-1} \begin{bmatrix} K_{\tilde{\mathbf{r}}1} & K_{\tilde{\mathbf{r}}\mathbf{r}} \end{bmatrix}. \quad (8)$$

Since α is the parent of 1 and \mathbf{r} , $\hat{P}_{1\tilde{1}}$ and $\hat{P}_{\mathbf{r}\tilde{\mathbf{r}}}$ have been already computed. $K_{\tilde{\mathbf{r}}1}\hat{P}_{\mathbf{r}\tilde{\mathbf{r}}}$ and $K_{\tilde{\mathbf{r}}1}\hat{P}_{1\tilde{1}}$ are computed by GEMM, and the reduced system is factorized by GETRF.

We can exploit a “telescoping” relation between $\hat{P}_{1\tilde{1}}$, $\hat{P}_{\mathbf{r}\tilde{\mathbf{r}}}$ and $\hat{P}_{\alpha\tilde{\alpha}}$. We say that $P_{\alpha\tilde{\alpha}}$ is “telescoped” from $P_{[\tilde{1}\tilde{\mathbf{r}}]\tilde{\alpha}}$, P_{11} , and $P_{\mathbf{r}\mathbf{r}}$ because is computed by formula in the box below.

$$D_\alpha^{-1} \boxed{P_{\alpha\tilde{\alpha}}} = \begin{bmatrix} \tilde{K}_{11}^{-1} & \\ & \tilde{K}_{\mathbf{r}\mathbf{r}}^{-1} \end{bmatrix} \boxed{\begin{bmatrix} P_{11} & \\ & P_{\mathbf{r}\mathbf{r}} \end{bmatrix} P_{[\tilde{1}\tilde{\mathbf{r}}]\tilde{\alpha}}}. \quad (9)$$

The calculation in the box requires just GEMM operations from the children (1 and \mathbf{r}) but not all descendants. Since $\hat{P}_{\alpha\tilde{\alpha}} = \tilde{K}_{\alpha\alpha}^{-1}P_{\alpha\tilde{\alpha}} = (I + W_\alpha V_\alpha)^{-1}D_\alpha^{-1}P_{\alpha\tilde{\alpha}}$, we can replace $D_\alpha^{-1}P_{\alpha\tilde{\alpha}}$ with (9). Now we find that $\hat{P}_{\alpha\tilde{\alpha}}$ can also be telescoped by $\hat{P}_{1\tilde{1}}$ and $\hat{P}_{\mathbf{r}\tilde{\mathbf{r}}}$ as

$$\hat{P}_{\alpha\tilde{\alpha}} = \left(I + \begin{bmatrix} \hat{P}_{1\tilde{1}} & \\ & \hat{P}_{\mathbf{r}\tilde{\mathbf{r}}} \end{bmatrix} V_\alpha \right)^{-1} \begin{bmatrix} \hat{P}_{1\tilde{1}} & \\ & \hat{P}_{\mathbf{r}\tilde{\mathbf{r}}} \end{bmatrix} P_{[\tilde{1}\tilde{\mathbf{r}}]\tilde{\alpha}}. \quad (10)$$

Notice that we no longer need to solve \tilde{K}_{11}^{-1} and $\tilde{K}_{\mathbf{r}\mathbf{r}}^{-1}$ in (10). Thus, no tree traversal is required. In the leaf level (base case), $\hat{P}_{\alpha\tilde{\alpha}}$ is computed directly from $K_{\alpha\alpha}^{-1}P_{\alpha\tilde{\alpha}}$.

Given these formulas and the skeletonization computed in Algorithm II.1, we compute the factors needed for the direct solver in a postorder traversal of the tree (Algorithm II.2). If α is a leaf node, we factorize $\lambda I + K_{\alpha\alpha}$ using an LU factorization. Otherwise, we compute $K_{\tilde{\mathbf{r}}1}$ and $K_{\tilde{\mathbf{r}}\mathbf{r}}$. Notice that $\hat{P}_{1\tilde{1}}$ and $\hat{P}_{\mathbf{r}\tilde{\mathbf{r}}}$ are computed in the previous recursion; thus, we can form and factorize the reduced system Z_α . Finally, $\hat{P}_{\alpha\tilde{\alpha}}$ is telescoped using (10), thus $\text{Solve}(\alpha, W_\alpha P_{[\tilde{1}\tilde{\mathbf{r}}]\tilde{\alpha}}, \text{false})$ (Algorithm II.3) will not invoke recursion.

This algorithm improves on the one in [36] by removing the extra subtree traversals that result in $\mathcal{O}(N \log^2 N)$ complexity. Instead, our algorithm exploits the nested structure of $\hat{P}_{\alpha\tilde{\alpha}}$ resulting in an $N \log N$ complexity for the factorization. In some of our largest runs, this resulted in over $3 \times$ speedup without any change in the accuracy.

Algorithm II.2 Factorize(α)

```

if  $\alpha$  is leaf then
  LU factorization  $\lambda I + K_{\alpha\alpha}$ .
   $W_\alpha = P_{\alpha\tilde{\alpha}}$  and  $P_{[\tilde{r}]\tilde{\alpha}} = I$ .
else
  Factorize(l) and Factorize(r).
  Form  $W_\alpha$  with  $\hat{P}_{1\tilde{r}}$ ,  $\hat{P}_{r\tilde{r}}$ , and  $V_\alpha$  with  $K_{1\tilde{r}}$ ,  $K_{r\tilde{r}}$ .
  LU factorize the reduced system  $Z_\alpha$  in (8).
   $\hat{P}_{\alpha\tilde{\alpha}} = \text{Solve}(\alpha, W_\alpha P_{[\tilde{r}]\tilde{\alpha}}, \text{false})$  using (10).

```

Algorithm II.3 $\underline{w} = \text{Solve}(\alpha, \underline{u}, \text{do_recur})$

```

if  $\alpha$  is leaf then LU solver  $\underline{w} = (\lambda I + K_{\alpha\alpha})^{-1} \underline{u}$ 
else
  if do_recur then
     $v = [\text{Solve}(l, \underline{u}_l, \text{true}); \text{Solve}(r, \underline{u}_r, \text{true})]$ .

  Compute  $\underline{w} = u - W_\alpha Z_\alpha^{-1} V_\alpha u$  using (8).

```

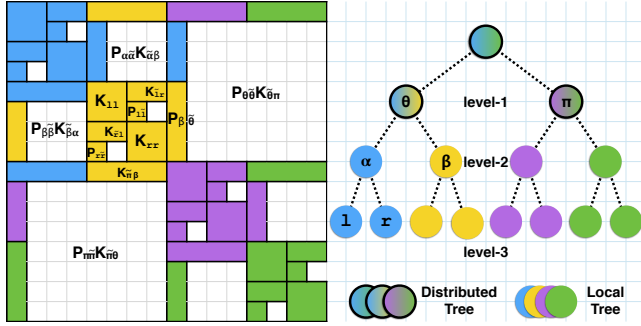


Figure 1 The top four levels of the tree and the corresponding blocks of the matrix \tilde{K} . The nodes belonging to each process are highlighted in a single color. Each process factorizes its own portion of the tree independently. We also highlight the factors used in the direct solver construction and show which process owns which factor. Each process owns a diagonal block and all factors in the same column and the same row. For example, the yellow process owns $P_{\beta\tilde{\beta}}$ and $K_{\alpha\beta}$ at level 1; similarly it owns $P_{\beta\tilde{\theta}}$ and $K_{\pi\beta}$ at level 0.

We then describe how to apply $\tilde{K}_{\alpha\alpha}^{-1}$ to a vector \underline{u} , shown in Algorithm II.3. If α is a leaf node, we can directly invoke an LU solver to obtain $\underline{w} = K_{\alpha\alpha}^{-1} \underline{u}$. Otherwise, we have two situations. If Algorithm II.3 is called by Factorize (do_recur is false), then we know $u = W_\alpha P_{[\tilde{r}]\tilde{\alpha}}$. Thus, no recursion is required. While Solve is called to solve a random u , then we need to solve $\tilde{K}_{11}^{-1} \underline{u}_1$ and $\tilde{K}_{rr}^{-1} \underline{u}_r$ recursively and compute (8) with GEMV on V_α and W_α and an LU solve GETRS on Z_α .

Therefore, the complete algorithm consists of constructing the tree, calling Algorithm II.1, then Algorithm II.2 and Algorithm II.3, each called on the root of the tree.

Parallel direct solver. The parallelization is essentially identical to the scheme proposed in [36]. Each subtree (a set of points $\{x\}$) is assigned to a distributed-memory process (or a worker). Although we described recursive version of our algorithms we use level-by-level traversals combined with shared or distributed memory parallelism (depending on the level) across nodes in the same level. If the number of nodes is less than the number of physical cores, the OpenMP

Algorithm II.4 DistFactorize(α, q)

```

if  $\alpha$  is at level  $\log p$  then Factorize( $\alpha$ ).
else
  DistFactorize(c,  $\frac{q}{2}$ ).

   $\{i < \frac{q}{2}\}$  —————  $\{i \geq \frac{q}{2}\}$  —————
   $\{0\}$  Send  $\tilde{l}$ .  $\{ \frac{q}{2} \}$  Recv, Bcast  $\tilde{l}$ .
   $\{0\}$  Recv, Bcast  $\tilde{r}$ .  $\{ \frac{q}{2} \}$  Send  $\tilde{r}$ .
  Reduce  $K_{\tilde{r}\{x\}} \hat{P}_{\{x\}\tilde{l}}$ . Reduce  $K_{\tilde{l}\{x\}} \hat{P}_{\{x\}\tilde{r}}$ .
   $\{0\}$  Recv  $\{ \frac{q}{2} \}$  Send  $K_{\tilde{l}\{x\}} \hat{P}_{\{x\}\tilde{r}}$ .

   $\{0\}$  Bcast  $P_{[\tilde{r}]\tilde{\alpha}}$  and LU factorizes  $Z$ .
   $\hat{P}_{\{x\}\tilde{\alpha}} = \text{DistSolve}(\alpha, \hat{P}_{\{x\}\tilde{r}} P_{[\tilde{r}]\tilde{\alpha}}, q, \text{false})$ .

```

Algorithm II.5 $\underline{w} = \text{DistSolve}(\alpha, \underline{u}, q, \text{do_recur})$

```

if  $\alpha$  is at level  $\log p$  then  $w = \text{Solve}(\alpha, \underline{u}, \text{do\_recur})$ .
else
  if do_recur then
     $u = \text{DistSolve}(c, \underline{u}, \frac{q}{2}, \text{do\_recur})$ .

     $\{i < \frac{q}{2}\}$  —————  $\{i \geq \frac{q}{2}\}$  —————
    Reduce  $K_{\tilde{r}\{x\}} u$ . Reduce  $u_{\tilde{l}} = K_{\tilde{l}\{x\}} u$ .
     $\{0\}$  Recv  $u_{\tilde{l}}$ .  $\{ \frac{q}{2} \}$  Send  $u_{\tilde{l}}$ .
     $\{0\}$   $[u_{\tilde{l}}; u_{\tilde{r}}] = Z^{-1}[u_{\tilde{l}}; u_{\tilde{r}}]$  using (8).
     $\{0\}$  Send  $u_{\tilde{r}}$ .  $\{ \frac{q}{2} \}$  Recv  $u_{\tilde{r}}$ .
     $\{0\}$  Bcast  $u_{\tilde{l}}$ .  $\{ \frac{q}{2} \}$  Bcast  $u_{\tilde{r}}$ .
     $w = u - \hat{P}_{\{x\}\tilde{l}} u_{\tilde{l}}$ .  $w = u - \hat{P}_{\{x\}\tilde{r}} u_{\tilde{r}}$ .

```

nested construct is enabled such that each thread will invoke parallel BLAS or LAPACK routines. If we have p processes, then above level $\log p$ of the tree, we have to communicate to compute factors, since the terms needed are distributed among processes. We use the Message Passing Interface (MPI) library for distributed memory communication.

In Figure 1, we summarize the distributed-memory algorithm. The four colors represent four different MPI ranks and the nodes they own. The tree on the right shows $l = 2$ treenodes are uniquely assigned to ranks, but treenodes with $l > 2$ are shared among ranks. To facilitate collective communication, each distributed treenode creates a local communicator, which equally divides the ranks of the parent. We use $\{i\}$ to denote the i_{th} MPI rank in the local communicator. Consider the communicator of α , which involves q ranks. Let c denote the child of α that $\{i\}$ owns. Then $c = 1$ if $\{i < \frac{q}{2}\}$. Otherwise, $c = r$. For a distributed node α , data points $\{x\}_i$ owned by $\{i\}$ are never required by other MPI processes, and $\mathcal{X}_\alpha = \mathcal{X}_1 \cup \mathcal{X}_r = (\cup_{i < \frac{q}{2}} \{x\}_i) \cup (\cup_{i \geq \frac{q}{2}} \{x\}_i)$. However, skeletons $\tilde{\alpha}$ and $P_{\tilde{\alpha}[\tilde{r}]}$ are only stored on $\{0\}$. When its sibling needs this information, we exchange the information using a SendRecv between $\{0\}$ and $\{ \frac{q}{2} \}$ using the parent communicator of α and its sibling. Once received, α and the sibling communicator can Bcast to every processes in their groups.

Algorithm II.4 describes the recursive distributed factorization. In each node α , ranks $\{i < \frac{q}{2}\}$ requires skele-

Arch	d	4	20	36	68	132	260
Haswell 16K	MKL+VML	31	53	72	115	190	305
	GSKS	321	465	512	634	687	680
KNL 16K	MKL+VML	12	93	132	416	636	916
	GSKS	703	888	1067	1246	1334	1449
Haswell 8K	MKL+VML	32	56	80	110	198	296
	GSKS	301	448	515	558	620	543
KNL 8K	MKL+VML	11	93	103	166	506	753
	GSKS	479	888	903	975	1220	1345
Haswell 4K	MKL+VML	30	52	70	110	180	284
	GSKS	250	359	384	420	477	468
KNL 4K	MKL+VML	11	56	76	116	370	578
	GSKS	341	445	464	510	858	1015

Table I Gaussian kernel summation efficiency of $16K \times 16K \times d$, $8K \times 8K \times d$, and $4K \times 4K \times d$ in GFLOPS. GSKS can be found in <https://github.com/ChenhanYu/ks>. The reference implementation uses MKL DGEMM and VML VEXP.

thus, an AllReduce is required at the end such that all MPI ranks get the same output. On the other hand, MatVecW is supposed to perform a scattering on $\{x\}$. Thus, MatVecW always happens after MatVecV. In this case, the inputs of Algorithm II.7 for all MPI ranks are the same.

D. Fast kernel summation

All the algorithms described in this paper rely on multiplying submatrices of K with vectors, which we refer to as “kernel summation”. These matrices can be precomputed and stored or they can be used in a matrix-free manner, by computing $K_{ij} = \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$ in $\mathcal{O}(d)$ time on the fly.

For example, during the factorization of K , submatrices $K_{\tilde{\beta}\alpha}$ can be computed and stored. In this case, MatVec of $K_{\tilde{\beta}\alpha}$ for all treenodes α in the solving phase can be done in $\mathcal{O}(sN \log N)$ with GEMV. However, storing all these submatrices requires $\mathcal{O}(sN \log N)$ memory. Memory requirements are even higher in the level-restricted version of our algorithm. Thus, we never store these submatrices in the hybrid methods due to the storage requirements.

Alternatively a matrix-free version only requires $\mathcal{O}(dN)$ storage (the coordinates of the points) and turns the GEMV to a GEMM. However, since kernel evaluations are quite expensive, this calculation can be significantly slower than storing the matrix and computing summation using GEMV. Our goal is to reduce the storage requirements without significantly sacrificing performance.

In [24], we presented GSKS (General Stride Kernel Summation), an matrix-free kernel summation that performs fusing optimization. While the best-known method computes

$$K_{\tilde{\beta}\alpha} u = \text{GEMV}(\mathcal{K}(\text{GEMM}(\mathcal{X}_{\tilde{\beta}}^T, \mathcal{X}_{\alpha})), u), \quad (11)$$

GSKS fuses \mathcal{K} (kernel function) and GEMV (reduction) into GEMM (semi-ring rank- d update). [35] uses the same idea to fuse nearest-neighbor search into GEMM. With a BLIS-like framework [30], matrix-matrix multiplication ($C = AB$) is divided into subproblems. A small subproblem that fits C into registers is implemented in vectorized assembly or intrinsic to maximize FLOPS throughput. The idea is to directly perform kernel evaluation and the GEMV on C while it is still in the register and only store back a vector w . In short, for a typical kernel summation that involves an $m \times n \times d$ GEMM with $\mathcal{O}(md + nd + mn)$ MOPS (Memory

Operations) in the best known method, GSKS can achieve $\mathcal{O}(mnd)$ FLOPS but with only $\mathcal{O}(md + nd)$ MOPS. This helps the computation become less memory bound even with small d . In this work, we implement this idea in AVX2 and AVX512 for Haswell and KNL architectures. We present the performance of these two different approaches in Table I. Due to the $\mathcal{O}(mn)$ memory saving, GSKS is about $3 \sim 30\times$ faster than the best known method on KNL for large problem size² and $d < 68$. We see that using GSKS significantly outperforms using the standard approach.

III. THEORY

Here, we present some theoretical complexity guarantees and discuss the stability of our direct solver.

Work. We present the complexity analysis of Algorithms II.2, II.3 and II.6. Throughout, we fix the leaf size m , level restriction L , and maximum skeleton size s . $T^f(N)$ denotes the complexity of Algorithm II.2, and $T^s(N)$ of Algorithm II.3, each for N points. Since Solve does either an LU solve or matrix-vector multiply in each step, we have

$$T^s(N) = 2T^s(N/2) + \mathcal{O}(Ns + s^2) = \mathcal{O}(sN \log N). \quad (12)$$

Notice that solving $\hat{P}_{\alpha\tilde{\alpha}} = \tilde{K}_{\alpha\alpha}^{-1} P_{\alpha\tilde{\alpha}}$ does not require traversing to the leaf level. Instead (10) only takes $\mathcal{O}(s^2 N)$ work. Using the complexity above, we derive $T^f(N)$ as

$$T^f(N) = 2T^f(N/2) + s^2 N + s^3 = \mathcal{O}(s^2 N \log N). \quad (13)$$

In the hybrid solver, each $(I + VW)x$ operation requires $\mathcal{O}(2^L s N)$ work. To summarize, both Algorithm II.2 and Algorithm II.3 take $\mathcal{O}(N \log N)$ work, and Algorithm II.6 takes $\mathcal{O}(N \log N)$ with additional $\mathcal{O}(N)$ for each iteration if L is independent from N .

Communication. The communication cost for the solving phase is $\mathcal{O}(s \log p)$ per level. To traverse the whole tree, $\mathcal{O}(s \log^2 p)$ is required per right hand side. However, during the factorization the solving phase does not recur. Thus, instead of $\mathcal{O}(s^2 \log^2 p)$ for s right hand sides, there is only $\mathcal{O}(s^2 \log p)$ communication per level. Overall, the communication cost for the full factorization is $\mathcal{O}(s^2 \log^2 p)$ since there are $\log p$ distributed levels.

Memory. The memory cost of our methods depend on level restriction L and maximum skeleton size s . These requirements are in addition to the cost to store the coordinates and skeleton information for ASKIT, reported in [21]. In our direct solver, we require the factors U , V , and $I + WV$ for each level of the tree below the level-restriction L in which skeletonization stops. This requires $\mathcal{O}(2sN + s^2)$ per level. Therefore, the overall memory required for our method is

$$\mathcal{O}\left((2sN + s^2) \left(\log\left(\frac{N}{m}\right) - L\right)\right). \quad (14)$$

Using GSKS can reduce $sN \log(N/m)$ to $\mathcal{O}(1)$ by computing V on the fly. Recomputing W with (10) can reduce another $sN \log(N/m)$ to sN . Using both schemes

²For small problem size, GEMM in LIBSMM <https://github.com/hfp/libsxmm> may be slightly faster than MKL GEMM, but it is still memory bound when d is small.

yields $\mathcal{O}(s^2(\log(N/m) - L) + sN)$ storage with $\mathcal{O}((d + s^2)N \log N)$ work (still $\mathcal{O}(N \log N)$ asymptotically).

Stability. Overall, the stability of our method is related to the conditioning of $(\lambda I + \tilde{K})$, D and the reduced system $(I + VW)$. We use $\kappa = \sigma_1/\sigma_{\max}$ to denote the 2-norm condition number of a matrix where σ_1 and σ_{\max} are the largest and smallest singular values of the matrix.

[10] suggests that when either U or V are orthonormal, then $\kappa(I + VW) \leq \kappa(D)\kappa(\tilde{K})$. Although, our U and V are not orthonormal, in our experience $\kappa(I + VW)$ does not have a conditioning problem when D and \tilde{K} are well-conditioned. The relation between $\kappa(\lambda I + D)$ and $\kappa(\lambda I + K)$ is more interesting. In general when h shrinks, we expect K to become more diagonally dominant and thus better conditioned. However, counter to this intuition, it is possible for D to become *more* poorly conditioned as h shrinks.

Since D is a submatrix of K , we have $\sigma_1(D) \leq \sigma_1(K)$ and $\sigma_n(D) \leq \sigma_n(K)$. When $\sigma_n(K) < \lambda$, then $\kappa(\lambda I + D) < \kappa(\lambda I + K)$ since λ dominates in the denominator. However when $\sigma_n(K) > \lambda$, $\kappa(\lambda I + D)$ can grow even as $\kappa(\lambda I + K)$ remains small. If this case happens in many levels of our factorization, then the method is not stable. With narrow bandwidths where K approaches a (blocked)-diagonal matrix, $\sigma_n > \lambda$ may occur.

Under the framework of hierarchical matrices, the pivoting rows we can choose during the D factorization are limited to the skeleton rows. Thus, even $\kappa(\lambda I + K)$ is not bad, $(\lambda I + D)$ can be unstable due to the aggressive pivoting strategy if λ is small. Our methods can detect this situation, but avoiding this case entirely (or fixing it) is not straightforward.

IV. EXPERIMENTAL SETUP

We performed numerical experiments on Haswell and KNL architectures with four different setups to examine the accuracy and efficiency of our methods. Especially, we want to demonstrate (1) the complexity improvement against [36], (2) FLOPS efficiency, (3) scalability and (4) the advantages of our hybrid solver. We explore the task of kernel ridge regression for binary supervised classification [33], which requires approximating the solution of $(\lambda I + K)^{-1}$ during the training step. We use the Gaussian kernel with bandwidth h . The model weight w is chosen by solving $w = (\lambda I + \tilde{K})^{-1}u$, where u is given (the labels). Once w is computed, the label given by $\underline{x} \notin \mathcal{X}$ is $\text{sign}(\mathcal{K}(\underline{x}, \mathcal{X})\underline{w})$. We apply our methods to train this model on real-world datasets employing up to 3,072 x86 cores and 4,352 KNL cores. The the percentage of correct predictions (Acc) is reported in Table II, along with the optimal h and λ that were found using holdout cross validation.

Implementation and hardware. Our experiments were conducted on Lonestar5 (two 12-core, 2.6GHz, Xeon E5-2690 v3 “Haswell” per node) and Stampede (68-core, 1.4GHz, Xeon Phi 7250 “KNL” per node) clusters at the Texas Advanced Computing Center. The theoretical

Dataset	N	d	h	λ	Acc
COVTYPE	0.1–0.5M	54	.07	.3	96%
SUSY	4.5M	8	.07	10	78%
MNIST2M	1.6M	784	.30	0	100%
HIGGS	10.5M	28	.90	.01	73%
MRI	3.2M	128	3.5	10	-
MNIST8M	8.1M	784	1.0	1.0	-
NORMAL	1–32M	64	.19	1.0	-

Table II Datasets used in the experiments. Here N denotes the size of the training set, and d is the dimensionality of points in the dataset. The testing sets are disjoint from the training sets. We sample 10K testing points and report the binary classification accuracy in the “Acc” column. h is the bandwidth of the Gaussian kernel used in our experiments. The regression results are produced using the parameters above. Some combinations we used during the cross-validation are presented in details in §V. **MRI**, **MNIST8M** and **NORMAL** are not used in regression tasks. For the **MNIST2M** we perform one-vs-all binary classification for the digit ‘3’. All coordinates are normalized to have zero mean and unit variance.

peak³ performance is 998 GFLOPS per Haswell node and 3,046 GFLOPS per KNL node. Inv-Askit and GSKS are compiled with intel-16 -O3 -mavx on Lonestar5 and intel-17 -O3 -xMIC-AVX512 on Stampede. All iterative solvers employ a Krylov subspace method (GMRES) from the PETSc library [3]. Specifically, we use modified Gram-Schmidt for re-orthogonalization and employ GMRES CGS refinement. If not specified, KNL experiments use Cache-Quadrant configuration with OMP_PROC_BIND=spread. “T” refers to the total runtime in seconds, and “GFs” refers to the GFLOPS per node.

Datasets. We use real-world datasets: **COVTYPE** (forest cartographic variables); **SUSY** and **HIGGS** (high-energy physics) [19]; **MNIST** (handwritten digit recognition) [5]; and **MRI** (brain MRI) [25]. We also use a 64D synthetic dataset, which is drawn from a 6D Normal distribution and embedded in 64D with additional noise. This set is a dataset with a high ambient but relatively small intrinsic dimension.

Accuracy metrics and parameter selection. For the linear solve, we report the relative residual

$$\epsilon_r = \|u - (\lambda I + \tilde{K})w\|_2 / \|u\|_2. \quad (15)$$

The parameters h and λ used in the Gaussian kernel were selected using cross-validation. In Table II we report the parameters we used. Other combinations in §V are candidates for the cross-validation. Level restriction L is chosen such that the relative error is controlled. In Table IV, V we use $L = 3$, and for experiments in Figure 5 we use $L = 5$ or 7.

V. EMPIRICAL RESULTS

The experiments are labeled #1 to #39 in the tables and figures. We select representative parameter combinations and compare the runtime between [36] and our Algorithm II.4 in Table III. We present single node performance on Haswell and KNL in Table IV. In Figure 4, we present strong scalability and verify the $\mathcal{O}(N \log N)$ complexity of our methods (Algorithm II.4). In Table V, we compare our

³We estimate the peak according to the clockrate and the theoretical FMA throughput. For 24 Haswell cores, $998 = 2 \times 12 \times 2.6 \times 16$. For 68 KNL cores, $3046 = 68 \times 1.4 \times 32$. As a reference, MKL GEMM can achieve 87% on the Haswell node and 69% on the KNL node. We assume two VPU can dual issue DFMA [29]. However, Intel processors may have a different frequency while fully issuing FMA, and the clockrate may drop to 1.0 GHz. This may be the reason why MKL GEMM can only achieve 2.1 TFLOPS on KNL.

τ			1E-1		1E-3		1E-5	
#	dataset	h	\log^2	\log	\log^2	\log	\log^2	\log
1	COVTYPE	.35	< 1	< 1	5	3	15	8
2		.07	5	3	24	11	27	12
3	SUSY	.50	< 1	< 1	1	< 1	6	3
4		.05	63	23	125	37	125	37
5	MNIST2M	1.0	24	11	38	15	40	16
6		.10	1	< 1	1	< 1	4	7
7	MNIST8M	0.3	142	51	185	61	195	64
8	HIGGS	2.0	75	29	324	84	326	85
9		.90	216	66	317	83	323	85
10	NORMAL32M	.19	18	10	46	22	84	28

Table III Factorization time comparison in second. Experiments are done on Lonestar5 using 128 nodes (3,072 cores) with adaptive ranks s selected by τ and s_{\max} . COVTYPE used $m = 2,048$, $\kappa = 2,048$, $s_{\max} = 2,048$. SUSY used the same combination. MNIST used $\kappa = 256$. HIGGS used $m = 512$, $\kappa = 1,024$. NORMAL used $m = 512$, $\kappa = 128$ and $s_{\max} = 256$.

hybrid (Algorithm II.6) with the direct method (Algorithm II.5). In Figure 5, we report the convergence behavior of iterative solver (Algorithm II.6) on $\lambda I + \tilde{K}$ and our hybrid factorization. Here the parameter τ indicates the relative tolerance of approximation of the kernel matrix K , s_{\max} is the maximum skeleton size, κ is the number of nearest neighbors used for skeletonization sampling in ASKIT, m is the leaf node size, and L is the level restriction.

Comparison with [36] (Table III). We compare the factorization time between the $\mathcal{O}(N \log^2 N)$ algorithm [36] and our $\mathcal{O}(N \log N)$ algorithms using the same parameters. We only compare the case without level restriction, because [36] does not support this feature. The runtime is directly associated with the rank s . For example, the U , V matrices in #4 are much larger than those in #3. Thus, the runtime is also much longer. The overall speedup is about 2–4 \times due to the $\log N$ term. Both methods construct exactly the same factorization (up to roundoff errors). Although the speedup is not exactly $\log N$ due to the prefactors depending on s and d , we can expect the asymptotic speedup to be $\mathcal{O}(\log N)$. For example, COVTYPE can only achieve 1.9 \times , but HIGGS can achieve 3.8 \times because the problem size is 20 \times larger. Both methods have $N \log N$ complexity for the “Solve” operation. For the experiments in Table III, the longest “Solve” operation (#8) is less than 2 seconds.

Single node performance (Table IV). We conduct a set of single node experiments to show the FLOP rates we achieve and to test some of the memory models on KNL. On a Haswell node, #11 reaches 62% (623/998) of the theoretical peak. For a KNL node, #13 (Cache-Quadrant) is the fastest and achieves 45% (1356/3046). Using MPI on KNL ($p > 1$) is typically slower due to the extra memory operations. Our implementation does not perform very well on the flat memory mode (#15 and 16). The memory requirements usually exceed 16GB and as a result U and V cannot fit into MCDRAM. We tried manually swapping memory between MCDRAM and DDR4 but this was not as efficient as using the cache memory mode.

Reducing storage (Table IV). In Table IV, we report three different schemes for kernel summation §II-D: GEMV, GEMM and GSKS. The first scheme takes $\mathcal{O}(sN \log N)$ time and space. The last two schemes evaluate $K_{\tilde{\beta}\alpha} u$ in

#	11	12	13	14	15	16
Config	Haswell		Cache-Quad		F-Quad	F-SNC4
p	1	4	1	4	1	4
nthd	24	6	68	17	68	17
T_f	89	94	41	52	62	77
GF_f	623	592	1356	1136	895	723
Compute MatVec V with GEMV.						
T_s	0.8	0.8	0.9	0.9	0.9	1.0
GF_s	14	14	12	12	12	11
Reevaluate V with GEMM.						
T_s	4.4	5.3	7.4	4.0	7.5	5.8
GF_s	158	119	40	74	39	51
Compute MatVec V with GSKS.						
T_s	1.1	1.8	1.1	1.2	1.2	1.3
GF_s	269	164	269	243	243	228

Table IV Single node performance on COVTYPE100K with $m = s = 2048$ (fixed rank) and $L = 3$. p is the number of MPI processes, and nthd is the number of OpenMP threads per process. #11 and #12 were conducted on Haswell, and #13 to #16 were conducted on KNL with different configurations. #13 and #14 use cache-quadrant, #15 uses flat-quadrant, and #16 uses flat-snc-4 configuration. #11 achieves the high factorization performance on Haswell, and #13 is the most efficient configuration on KNL. In each column, we report factorization time (T_f) and GFLOPS (GF_f) and three different solving time (T_s) and GFLOPS (GF_s) and three different solving time (T_s). These three solving schemes have different storage requirement and memory operations.

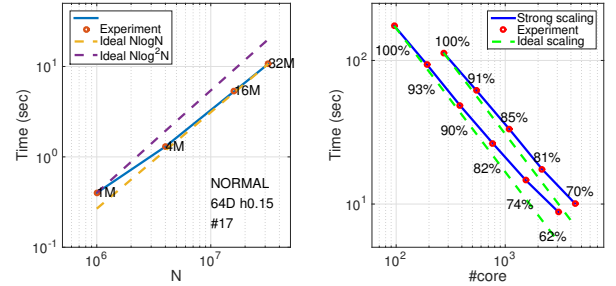


Figure 4 $\mathcal{O}(N \log N)$ verification (#17) and strong scaling (#18). #17 use NORMAL with $m = 512$, $\kappa = 128$, a fixed $s = 256$ and $L = 1$. The blue lines are experimental factorization time, and yellow lines are theoretical (ideal) time. #18 use NORMAL1M with $\kappa = 128$, $m = s = 2048$ and $L = 1$. We increase number of nodes up to 128 Haswell nodes (3,072 cores) and 64 KNL nodes (4,352 cores). The green lines represent the ideal scaling.

$\mathcal{O}(dsN \log N)$ where d is 54 in this dataset. GEMM takes $\mathcal{O}(sN)$ to store the matrix, but GSKS is matrix free with $\mathcal{O}(1)$ space. GSKS is only 1.2–1.6 \times slower than the GEMV approach while it is 4–7 \times faster than the GEMM approach. Notice that 80% of T_s is dominated by GETRS (triangular solver) when d is small. Since GETRS can only achieve 2 GFLOPS on a KNL node, the overall GF_s is somewhat low.

Scaling (Figure 4). In #17, we use 128 nodes (3,072 cores) and increase N from 1M to 32M. We can observe that our implementation is very close to the theoretical $N \log N$ scaling (yellow) but lower than the $N \log^2 N$ scaling (purple). In #18, we fix the data set (NORMAL 1M) and increase the number of cores. The green line is the ideal scaling (100%), and our implementation reaches 62% efficiency on 3,072 Haswell cores and 70% on 4,352 KNL cores. This relatively small problem (1M) cannot fully exploit all computing resources; thus, we can see the degradation while N is small or when the number of cores is large (~ 230 points per core for 64 KNL nodes).

Hybrid and direct methods comparison (Table V). In Table V we set $L = 3$ (level restriction) and compare Algorithm II.6 (hybrid) and Algorithm II.2 (direct) for problems that Algorithm II.2 can be applied (i.e., the

#	KNL	ASKIT	T_f	GF_f	T_s	GF_s	ϵ_r	KSP
SUSY $h = 0.15 \lambda = 40$								
19	x	294	60	844	1.5	82	5e-12	-
20	-	570	110	467	1.2	99	5e-12	-
21	-	544	59	414	22.3	246	6e-4	98
MRI $h = 3.5 \lambda = 10$								
22	x	302	46	795	1.4	319	1e-10	-
23	-	396	84	427	1.5	298	1e-10	-
24	-	217	37	467	39.6	508	3e-4	93
MNIST2M $h = 1.0 \lambda = 1$								
25	x	237	27	655	1.9	592	1e-13	-
26	-	270	47	372	2.3	489	1e-13	-
27	-	217	19	404	27.2	612	1e-3	27

Table V Hybrid and direct methods comparison with level restriction $L = 3$. All experiments use adaptive ranks with tolerance $\tau = 0.00001$ and $s_{\max} = 2048$. We report ASKIT building time, factorization time (T_f), solving time employing GSKS (T_s) and efficiency in GFLOPS. The three experiments with orange index are the hybrid scheme and the remaining are the direct factorization. We report the relative residual ϵ_r and number of Krylov iterations (KSP) (for the hybrid method).

full factorization requires $2^L s N + 2^{2L} s^2$ memory for level restriction L), with adaptive s selection. #20, #23 and #26 use the direct factorization on Haswell, and #19, #22, and #25 on KNL. The remaining three runs are done on Haswell using the hybrid method. On Haswell, we observe that the factorization time T_f is about two times longer than #21, #24 and #27. In the factorization phase, both Haswell and KNL do not perform as well as in Table IV, because using adaptive ranks s results in load imbalance. If we further increase L , as we need to do in Figure 5, the cost of the full factorization can be $1000\times$ in runtime and $30\times$ in storage.

Applying the hybrid solver to a vector is slower than applying the direct solver, due to the need of iteration. E.g., T_s in #21 is about $20\times$ slower. Yet the overall runtime ($T_f + T_s$) of the hybrid method is still smaller than the direct one. When L is larger, the advantage of the hybrid solver will be higher. For example, in Figure 5, we report results that require $L = 7$. Algorithm II.2 cannot be used: the memory just for Z with $s = 2048$ exceeds 500GB.

Convergence behavior for solving $\lambda I + \tilde{K}$ (Figure 5). We report the convergence rate using four different bandwidths with two different methods: (a) unpreconditioned GMRES using ASKIT’s MatVec for $\lambda I + \tilde{K}$ (blue line) and (b) our hybrid method Algorithm II.6 (orange line). Each row corresponds to a dataset with a specific h . These experiments resemble a cross-validation study in which we vary λ in order to improve learning. Across columns, we vary λ as $[10^{-2}, 10^{-3}, 10^{-5}]_{\sigma_1(\tilde{K})}$, where $\sigma_1(\tilde{K})$ is an estimate, so that the condition number κ of $\lambda I + \tilde{K}$ is 10^2 , 10^3 and 10^5 respectively. We report the relative (to a zero initial guess) Krylov residual ϵ_r (y-axis) over time (x-axis). The steeper the curve is, the faster the method converges. The x-axis offset represents setup costs. E.g., in #28, the offset of the blue line (≈ 140 sec) is the cost of building the tree and the skeletons (spent in ASKIT). The fixed cost of (b) includes the fixed cost of (a) plus the factorization time.

We see that most of the blue lines are flat when the condition number is around $1E+5$, but orange lines still decrease steadily except for #30 (see §III for the stability issue). We can observe $10\text{--}1000\times$ speedup on the “Solve” operations. Overall the hybrid scheme is faster and has

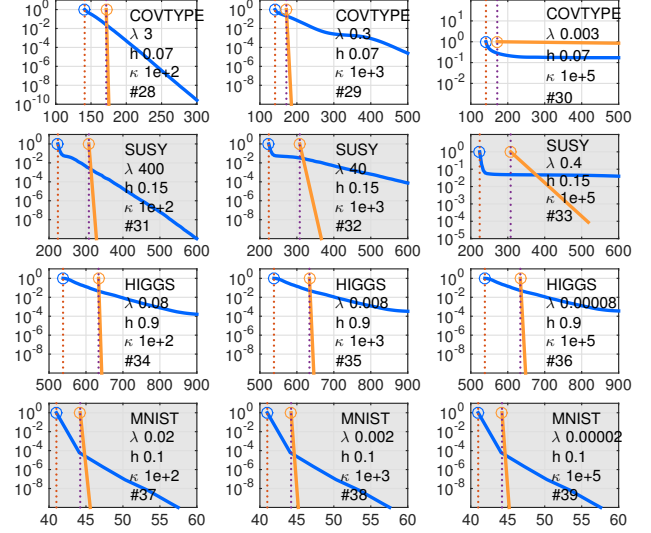


Figure 5 solving $\lambda I + \tilde{K}$: Convergence of the relative residual ϵ_r (vertical axis) over time (horizontal axis) in seconds. (a) (blue) is unpreconditioned GMRES; (b) (orange) is hybrid method. All experiments used $\tau = 1E - 5$, m , k and s_{\max} are the same as Table III; κ is the condition number. COVTYPE and HIGGS used $L = 5$ restriction. SUSY and MNIST used $L = 7$. These runs may be $100\text{--}1000\times$ more expensive (also running out of memory) if we were to use the direct solver.

more predictable behavior. #30 is detected numerically ill-conditioning of D in our solver. Also notice that in #30 both methods fail to converge.

VI. CONCLUSIONS

We have introduced new algorithms for approximately factorizing kernel matrices. We evaluated our algorithms on both real-world and synthetic datasets with different parameters. We conducted analysis and experiments to study the complexity and the scalability of our methods. These experiments include scaling up to 3,072 Haswell cores and 4,352 KNL and exhibit significant speedups over existing methods. The factorization can be very fast. For example, it only takes 10 seconds to factorize a kernel matrix with 32M points in 64D. Our future work will focus on further optimization of our implementation. In particular, we would like to introduce task parallelism in the tree traversal to address the load balancing issue. While adaptive ranks or adaptive level restriction is used, each treenode may have different workload. In this case, scheduling is important to avoid the critical path. Additionally, we plan to address the stability issues mentioned in §III and explore other possible variants (e.g. sparse off-diagonal blocks).

REFERENCES

- [1] S. AMBIKASARAN AND E. DARVE, *An $\mathcal{O}(n \log n)$ fast direct solver for partial hierarchically semi-separable matrices*, Journal of Scientific Computing, 57 (2013), pp. 477–501.
- [2] S. AMBIKASARAN, D. FOREMAN-MACKEY, L. GREENGARD, D. W. HOGG, AND M. O’NEIL, *Fast direct methods for Gaussian processes and the analysis of NASA Kepler mission data*, arXiv preprint arXiv:1403.6015, (2014).
- [3] S. BALAY ET AL., *PETSc Web page*, 2014.
- [4] M. BEBENDORF, *Hierarchical matrices*, Springer, 2008.
- [5] C.-C. CHANG AND C.-J. LIN, *Libsvm: A library for support vector machines*, ACM Transactions on Intelligent Systems and Technology (TIST), 2 (2011), p. 27.
- [6] B. DAI, B. XIE, N. HE, Y. LIANG, A. RAJ, M.-F. F. BALCAN, AND L. SONG, *Scalable kernel methods via doubly stochastic gradients*, in Advances in Neural Information Processing Systems, 2014, pp. 3041–3049.
- [7] A. GITTENS AND M. MAHONEY, *Revisiting the Nystrom method for improved large-scale machine learning*, in Proceedings of ICML13, 2013, pp. 567–575.
- [8] A. GRAY AND A. MOORE, *N-body problems in statistical learning*, Advances in neural information processing systems, (2001), pp. 521–527.
- [9] W. HACKBUSCH, B. N. KHOROMSKII, AND E. E. TYR- TYSHNIKOV, *Approximate iterations for structured matrices*, Numerische Mathematik, 109 (2008), pp. 365–383.
- [10] W. W. HAGER, *Updating the inverse of a matrix*, SIAM review, 31 (1989), pp. 221–239.
- [11] N. HALKO, P.-G. MARTINSSON, AND J. TROPP, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review, 53 (2011), pp. 217–288.
- [12] T. HOFMANN, B. SCHÖLKOPF, AND A. J. SMOLA, *Kernel methods in machine learning*, The annals of statistics, (2008), pp. 1171–1220.
- [13] C.-J. HSIEH, S. SI, AND I. S. DHILLON, *Fast prediction for large-scale kernel machines*, in Advances in Neural Information Processing Systems 27, 2014, pp. 3689–3697.
- [14] M. IZADI KHALEGHABADI ET AL., *Parallel H-matrix arithmetic on distributed-memory systems*, (2012).
- [15] R. KONDOR, N. TENEVA, AND V. GARG, *Multiresolution matrix factorization*, in Proceedings of ICML14, 2014, pp. 1620–1628.
- [16] R. KRIEMANN, *Parallele Algorithmen fr H-Matrizen*, dissertation, Universitt Kiel, 2004.
- [17] ———, *H-LU factorization on many-core systems*, Preprint, Max Planck Institute for Mathematics in the Sciences, 2014.
- [18] D. LEE, A. GRAY, AND A. MOORE, *Dual-tree fast gauss transforms*, Advances in Neural Information Processing Systems, 18 (2006), pp. 747–754.
- [19] M. LICHMAN, *UCI machine learning repository*, 2013.
- [20] Z. LU ET AL., *How to scale up kernel methods to be as good as deep neural nets*, arXiv preprint arXiv:1411.4000, (2014).
- [21] W. B. MARCH, B. XIAO, AND G. BIROS, *ASKIT: Approximate skeletonization kernel-independent treecode in high dimensions*, SIAM Journal on Scientific Computing, 37 (2015), pp. 1089–1110.
- [22] W. B. MARCH, B. XIAO, S. THARAKAN, C. D. YU, AND G. BIROS, *A kernel-independent FMM in general dimensions*, in Proceedings of SC15, Austin, Texas, Nov 2015.
- [23] ———, *Robust treecode approximation for kernel machines*, in Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Sydney, Australia, August 2015, pp. 1–10.
- [24] W. B. MARCH, B. XIAO, C. D. YU, AND G. BIROS, *An algebraic parallel treecode in arbitrary dimensions*, in Proceedings of IPDPS15, Hyderabad, India, May 2015.
- [25] MENZE ET AL., *The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS)*, IEEE Transactions on Medical Imaging, (2014), p. 33.
- [26] S. M. OMOHUNDRO, *Five balltree construction algorithms*, International Computer Science Institute Berkeley, 1989.
- [27] C. E. RASMUSSEN AND C. WILLIAMS, *Gaussian Processes for Machine Learning*, MIT Press, 2006.
- [28] S. SI, C.-J. HSIEH, AND I. DHILLON, *Memory efficient kernel approximation*, in Proceedings of The 31st International Conference on Machine Learning, 2014, pp. 701–709.
- [29] A. SODANI ET AL., *Knights landing: Second-generation intel xeon phi product*, IEEE Micro, 36 (2016), pp. 34–46.
- [30] F. G. VAN ZEE AND R. A. VAN DE GEIJN, *Blis: A framework for rapidly instantiating blas functionality*, ACM Transactions on Mathematical Software (TOMS), 41 (2015), p. 14.
- [31] R. WANG, Y. LI, M. W. MAHONEY, AND E. DARVE, *Structured block basis factorization for scalable kernel matrix evaluation*, arXiv preprint arXiv:1505.00398, (2015).
- [32] S. WANG, X. S. LI, J. XIA, Y. SITU, AND M. V. DE HOOP, *Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures*, SIAM Journal on Scientific Computing, 35 (2013), pp. C519–C544.
- [33] L. WASSERMAN, *All of Statistics: A Concise Course in Statistical Inference*, Springer, 2004.
- [34] C. WILLIAMS AND M. SEEGER, *Using the Nyström method to speed up kernel machines*, in Proceedings of the 14th Annual Conference on Neural Information Processing Systems, no. EPFL-CONF-161322, 2001, pp. 682–688.
- [35] C. D. YU, J. HUANG, W. AUSTIN, B. XIAO, AND G. BIROS, *Performance optimization for the k-nearest neighbors kernel on x86 architectures*, in Proceedings of SC15, ACM, 2015, pp. 7:1–7:12.
- [36] C. D. YU, W. B. MARCH, B. XIAO, AND G. BIROS, *INV-ASKIT: a parallel fast direct solver for kernel matrices*, in Proceedings of the IPDPS16, Chicago, USA, May 2016.